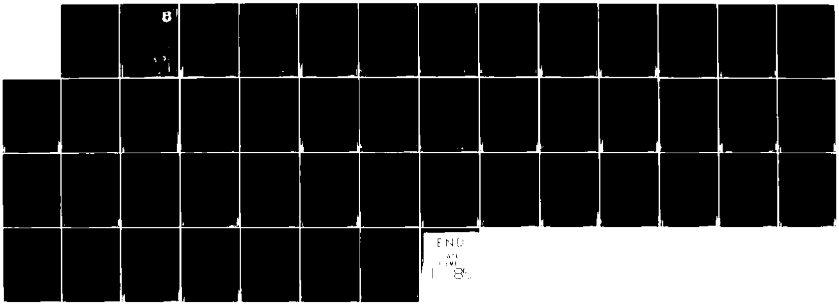
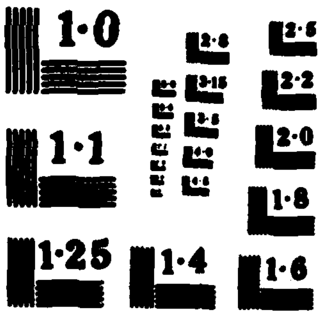
AD A147 827 A SIMULATOR FOR DETERMINING THE PERFORMANCE OF TRANSACTION MANAGER AND LO..(U) ELECTRONIC SYSTEMS DIV HANSCOM AFB MA D A VARVEL ET AL. 31 AUG 84 1/1

UNCLASSIFIED ESD TR-84-195 F/G 9/2 NL



END

1 85


1·0
2·8
2·5
3·15
2·2
3·6
1·1
4·0
2·0
4·5
1·8
1·25
1·4
1·6

(12)

ESD-TR-84-195

A Simulator for Determining the Performance of Transaction Manager and Lock Manager Combinations in a Database

DONALD A. VARVEL
WILLIAM PERRIZO

31 August 1984

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED

Prepared for

ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
DEPUTY FOR DEVELOPMENT PLANS
HANSCOM AIR FORCE BASE, MASSACHUSETTS 01731

REVIEW AND APPROVAL

This technical report has been reviewed and is approved for publication.

[illegible]
THOMAS SELINKA, Capt. USAF
Project Officer

[illegible]
SILVIO V. D'ARCO, Lt Col. USAF
Deputy Director, Tactical C^3I Systems Planning
Deputy for Development Plans

FOR THE COMMANDER

[illegible]
DONALD L. MILLER, Colonel, USAF
Assistant Deputy for Development Plans

AD-A147827

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) ESD-TR-84-195	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Hq Electronic Systems Division Deputy for Development Plans 6b. OFFICE SYMBOL (If applicable) ESD/XR	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code) Hanscom AFB Bedford, MA 01731	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT NO.
11. TITLE (Include Security Classification) A Simulator for Determining the (Cont.)	

12. PERSONAL AUTHOR(S)
Donald A. Varvel and William Perrizo

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Final	FROM ______ TO ______	1984 August 31	50

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Blocking, Concurrency control, Database Management System, DBMS, Design, Locking, Modularization, Pascal, Performance. Tactical Air Control System, Transactions.

Cont

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Most database systems use locking for concurrency control. Responsiveness is degraded when transactions spend much time waiting for locks. In those situations in which the lockable units need not be processed in a particular order, differences in the order of processing can make large differences in the durations of the transactions, i.e., responsiveness. Order of processing may be modified by the use of a combination of non-blocking and potentially-blocking lock requests. A simulation is used to investigate the performance of several such algorithms in a variety of settings. Originator-supplied Keywords include:

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☐ [✓ SAME AS RPT.] DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Capt [illegible]	22b. TELEPHONE NUMBER (Include Area Code) (617) 271-8400 22c. OFFICE SYMBOL [illegible]

DD FORM 1473, 83 APR   EDITION OF 1 JAN 73 IS OBSOLETE.   Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

Block 11 Continued

Performance of Transaction Manager and Lock Manager Combinations in a Database. (Unclassified)

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

TABLE OF CONTENTS

TABLE OF CONTENTS

1. Introduction

The information system supporting distributed command and control (C^2) operations may be viewed as a distributed database. This is an attractive view because the distributed database literature includes solutions to some of the problems facing the implementor of a distributed C^2 operation. General discussions of distributed database may be found in [CERI84] and [DATE83], while methods applicable to C^2 site initialization, recovery, and back-up are discussed in [ATTA83].

Database techniques have been developed largely for a commercial environment where responsiveness is less important than complete accuracy, and short breaks in service may be tolerated. For example, concurrency control may involve locking units of information for extended periods of time, so that the information is not generally available. If transactions are kept short, as recommended in [DATE83], the disruption is not great. [ATTA83], however, outlines solutions to the problems of initialization and recovery that necessarily involve huge transactions.

We have therefore addressed the problem of responsiveness in a locking environment.

Many database transactions, and parts of nearly all, can be viewed as a set operations; this is particularly explicit in the relational model [CODD70]. Sets need not be accessed in any particular order, but on a sequential machine some order must be chosen. Usually that order is arbitrary. We propose to modify that order to increase responsiveness.

The order of access to data items becomes important in the event of conflict. Conflict occurs when one transaction attempts to access a data item on which another transaction holds a lock that is not compatible with the attempted access. In that case, the requesting transaction enters a nonbusy wait state. Barring deadlock it will return to the ready state when the requested lock becomes available.

Deadlock occurs when T_1 waits for a resource held by T_2 and T_2 waits, possibly indirectly, for a resource held by T_1. The situation in which the requesting transaction either receives a lock or waits assumes some means of deadlock detection and recovery. Our simulation assumes such a system, but we do not model the actual effects of deadlock.

Transaction T_1 must access five lockable units of data, U_1-U_5. Each lockable unit requires 10 time units to process. Assume that locks for U_1 and U_3-U_5 are available, and that the lock request for each requires one time unit. U_2, however, is locked and will remain locked until 50 time units after T_1 starts. Accessing the lockable [illegible]ts in the natural order, U_1, U_2, U_3, U_4, U_5, proceeds as in Table 1.

Activity	Time Required	Time When Complete
Lock U_1	1	1
Process U_1	10	11
Lock U_2	39	50
Process U_2	10	60
Lock U_3	1	61
Process U_3	10	71
Lock U_4	1	72
Process U_4	10	82
Lock U_5	1	83
Process U_5	10	93

Table 1

The alternative order U_1, U_3, U_4, U_5, U_2 proceeds as in Table 2.

Activity	Time Required	Time When Complete
Lock U_1	1	1
Process U_1	10	11
Lock U_3	1	12
Process U_3	10	22
Lock U_4	1	23
Process U_4	10	33
Lock U_5	1	34
Process U_5	10	44
Lock U_2	6	50
Process U_2	10	60

Table 2

This example shows that even in a very simple case the order of processing can have a substantial effect on the amount of real time required to process a given transaction. We have devised several means of improving the order of access to data items to avoid long waits caused by locking.

In Section 2 we outline three new lock managers and several algorithms for using them. In section 3 we describe and justify a simulation of combinations of lock managers and protocols from Section 2. Section 4 consists of a summary and conclusions. The simulation program is included as Appendix A, and the results as Appendix B.

2. Algorithms

In this section we present some new lock managers and the transaction managers that use them. First we define the context in which we will operate.

We will assume, for the sake o this discussion, a centralized (non-distributed) database that uses locks and deadlock detection.

A concurrent database system must have lock manager, which we abbreviate LM. LM0 below is an example. Transactions get access to data by requesting locks from the lock manager, and return the resources to the system by informing the LM of the release. The lock manager may cause transactions to become blocked or unblocked. It maintains queues for those data items that are requested while locked.

The acquisition portion of the usual lock manager may be viewed as the following:

```
Procedure LM0(Lockable_Unit_ID, Lock_Level, Transaction_ID);
Begin
   If Lockable_Unit_ID is available at Lock_Level then
      Record lock for Transaction_ID
   Else begin
      Place Transaction_ID in queue for Lockable_Unit_ID;
      Block Transaction_ID;
      Cause system to dispatch another transaction
   End
End;
```

A transaction manager (TM) is an entity that interprets high-level queries by issuing lower-level requests, including negotiations with the lock manager. Each transaction has a logical transaction manager, although the actual code might be shared. TMO is an example.

Here is how the usual transaction manager deals with a series of data items $D_1, \ldots, D_n$, which may be processed in any order:

```
Transaction manager TMO:
Begin
   For i := 1 to n do begin
      LMO($D_i$, Lock_Level, Trans_ID);
      Process $D_i$
   End
End;
```

A given transaction running on a given database at a particular time will find certain data items (which we will refer to as lockable units or LU's) available at certain times but not at others. In our simulation we present this as a fixed background. Queues of fairly stable length tend to form waiting for high-activity data items. Our simulation includes both lockable units with conflicts at certain times and LU's with fixed-length queues. We also simulate processing delays for the various lockable units.

The time required to acquire an available lock is taken as the unit. This lock-request delay is chosen as (relatively) large as it is on the assumption that the lock manager will at least occasionally have an entry queue.

LMO is not useful in determining whether or not to process an LU immediately. A more useful lock manager, LM1, would have two lock-request entry points: one blocking and one non-blocking. The blocking request is as LMO. The non-blocking request returns True or

False, depending on whether the lock is presently available. This is LM1, assuming LM0 is still available as a blocking entry point:

```
Function LM1(Lockable_Unit_ID, Lock_Level, Transaction_ID): Boolean;
Begin
   If Lockable_Unit_ID is available at Lock_Level then begin
      Record lock for Transaction_ID;
      LM1 := True
   End
   Else LM1 := False
End;
```

In LM2, if the lock is not available, the transaction is entered into its queue. This requires slight changes in the blocking and release parts of the lock manager: the transaction at the head of the queue may not be blocked, and the requesting transaction may already have the lock.

```
Function LM2(Lockable_Unit_ID, Lock_Level, Transaction_ID): Boolean;
Begin
   If Lockable_Unit_ID is already locked by Transaction_ID then
      LM2 := True
   Else if Lockable_Unit_ID is available at Lock_Level then begin
      Record lock for Transaction_ID;
      LM2 := True
   End
   Else begin
      If Transaction_ID is not in Lockable_Unit_ID's queue then
         Place Transaction_ID in Lockable_Unit_ID's queue;
      LM2 := False
   End
End;
```

It is possible to propose a variety of algorithms that use LM1 or LM2. We define and simulate three such algorithms.

Our first TM makes blocking requests only when an entire pass through the LU's using non-blocking requests produces no results. It may be used with either LM1 or LM2 (as may TM2 and TM3).

```
Transaction manager TM1:
Begin
```

```
      While unprocessed units remain do begin
         Repeat process list using nonblocking requests
         Until a pass finds all unprocessed units locked;
         If at least one unprocessed unit remains then
            Make a blocking request for an unprocessed unit
      End
   End;
```

The second TM makes a pass through the list of LU's using nonblocking requests and then arbitrarily selects an unprocessed LU to wait for: these two actions are alternated until no LU's remain unprocessed. It differs from TM1 in that is does not wait for an entire unsuccessful pass before issuing a blocking request.

```
   Transaction manager TM2:
   Begin
      While unprocessed LU's remain do begin
         Process list of LU's using nonblocking requests;
         If at least one unprocessed LU remains then
            Pick an unprocessed LU and make a blocking request for it
      End
   End;
```

TM3 makes still fewer nonblocking requests. It makes a nonblocking pass and then a blocking pass. It takes what is immediately available, than waits for what is not.

```
   Transaction manager TM3:
   Begin
      Process list of LU's using nonblocking requests;
      Process list of LU's using blocking requests
   End;
```

Transaction managers TM1, TM2, and TM3 all eventually choose arbitrary lockable units for which to issue blocking lock requests; in the simulation it is the first unprocessed LU. The one arbitrarily chosen may not be the best. Another LU may become available much earlier. We would like always to be sure of selecting the best one. Using the lock managers discussed so far, that cannot be done.

To do so we must make more substantial modifications to the lock manager. We need a lock manager that can be given a list of requests and instructed to unblock the transaction and return the lockable unit's ID when <u>any</u> of the lockable units become available.

LM3 has three parts: LM3I (<u>I</u>nitial) for nonblocking requests, LM3W (<u>W</u>ait) for multiple blocking requests, and LM3R (<u>R</u>elease) for releasing locks. (Each of the previous lock managers also must have a release portion, but all have been so similar to LM0's that we have omitted discussing them.) LM3 generates the blocking request list in LM3I.

```
Function LM3I(Lockable_Unit_Id,Lock_Level,Transaction_ID):Boolean;
Begin
   If Lockable_Unit_ID is already held by Transaction_ID then
      LM3I := True
   Else if Lockable_Unit_ID is available at Lock_Level then begin
      Record lock for Transaction_ID;
      LM3I := True
   End
   Else begin
      If Transaction_ID is not in Lockable_Unit_ID's queue then
         Place Transaction_ID in Lockable_Unit_ID's queue;
      If Or_list for Transaction_ID does not exist then
         Create an Or_list for Transaction_ID;
      Insert Lockable_Unit_ID in Or_list;
      LM3I := False
   End
End;


Function LM3W(Transaction_ID) : Lockable_Unit_ID_type;
Begin
   If Or_list is empty or non-existent then LM3W := error
   Else if no lock in the Or_list is available then begin
      Block Transaction_ID;
      Dispatch another transaction
   End
   Else begin
      Select an awarded lock;
      Remove that lock from the Or_list;
      LM3W := Lockable_Unit_ID
   End
End;
```

```
Procedure LM3R(Lockable_Unit_ID);
Begin
   If Lockable_Unit_ID's queue is nonempty then begin
      Award the lock to a transaction from the queue;
      If transaction is waiting then begin
         Remove that lock from the Or_list;
         Place the Lockable_Unit_ID where the transaction expects a
            return value from LM3W;
         Unblock the transaction
      End
   End
End;
```

```
Transaction manager TM4:
Begin
   Make a pass through the list using LM3I;
   While at least one unprocessed unit remains do
      Process(LM3W(Transaction_ID))
End;
```

3 The simulation

In this section we describe our simulation.

An estimate of performance will help determine the relative worth of the various methods. We chose to perform a Monte Carlo simulation in order to produce reasonable answers in reasonable time.

Transactions are generated at random within supplied parameters. A simulation of each Transaction Manager/Lock Manager combination is run against each generated transaction. Because the algorithms run against the same transactions, the results are comparable. The cycle of transaction generation and running of algorithms is repeated -- here, twenty times -- and the results averaged.

We have not modeled deadlock. Some authors [KUNG81, GRAY81] have maintained that deadlock is rare. In any case, we maintain that deadlock would not substantially change the relative results.

We wished to have a standard against which to compare each of the methods, including the usual TMO. We have defined an optimal time and have computed it for each transaction. We have compared each of the algorithms to that standard.

For information on the computation of optimal time, see [VARV84].

The two statistics we record are time active and lock requests. Minimum lock requests is achieved by TMO. The other transaction/lock manager combinations reduce time active at the cost of some additional lock requests.

We assume that a few extra lock requests may be tolerated rather well, but that a doubling of lock requests should purchase very substantial reductions of time active. Accordingly, our evaluation function is

$$(1 + (1 - R/U)^2)\ T$$

where T is time active, R is number of lock requests, and U is lockable units.

To put these numbers in perspective, we have displayed the ratio of each method's evaluation to optimal.

Each simulation determines the mean behavior of each of the algorithms running twenty transactions. Twelve simulations were run, representing the cross product of 5, 10, 15, and 20 lockable units with light, medium, and heavy activity.

The two probabilities in the setup parameters are not comparable. The second is each lockable unit's probability of having a queue. The first, however, is the probability of an adverse request in a given ten-time-unit interval. We used those two numbers to specify level of activity.

The results of the simulation are given in Appendix B.

4 Summary and conclusions

A transaction may wait for one locked unit while it could be processing others that are not locked. Worse, some of the other units could become locked in the interim. We have presented several approaches to solving that problem.

We have devised three modified lock managers, which we call LM1, LM2, and LM3. LM1 and LM2 may be used with any of the transaction managers TM1, TM2, and TM3, but LM3 is used only with TM4; thus, we present seven combinations.

The performance of these combinations has been simulated and compared with optimal performance. Each combination was tested with transactions of various sizes and with several levels of conflict.

4.1 Conclusions

No combination of transaction and lock managers has enough information to achieve optimal performance. We were able to simulate optimality only because all conflicts were known in advance. TM4/LM3 achieved the best performance in our simulation, but might prove difficult to implement. TM3/LM2 performed almost as well as TM4/LM3 and might constitute a good practical choice.

We believe that either TM3/LM2 or TM4/LM3 should be considered for implementation in database systems where response time is critical.

BIBLIOGRAPHY

[ATTA83]: Attar, R., Bernstein, P.A., and Goodman, N. "Site Initialization, Recovery and Back-up in a Distributed Database System," RADC-TR-83-226 Vol. I, October 1983. (ADA138891)

[BERN81]: Bernstein, P.A., and Goodman, N. "Concurrency Control in Distributed Database Systems," ACM Computing Surveys 13,2 (June, 1981), 185-221.

[CERI84]: Ceri, S., and Pelagatti, G. _Distributed Databases_, McGraw-Hill, 1984.

[CODD70]: Codd, E.F. "A Relational Model for Large Shared Data Banks," CACM 13,6 (June, 1970), 377-387.

[DATE83]: Date, C.J. _An Introduction to Database Systems, vol. II_, Addison-Wesley, 1983.

[ESWA76]: Eswaren, K.P., Gray, J.N., Lorie, R.A., and Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Data Base System," CACM 19,11 (November, 1976).

[GRAY81]: Gray, J.N., Homan, P., Korth, H., and Obermarck, R. "A Straw Man Analysis of the Probability of Waiting and Deadlock," IBM Research Report RJ3066 (February 1981).

[KUNG81]: Kung, H.T., and Robinson, J.T. "On Optimistic Methods for Concurrency Control," ACM TODS 6, 2 (June 1981).

[VARV84]: Varvel, D.A., and Perrizo, W. "Efficient Computation of Optimal Time for Transaction Processing in a Database System," ESD-TR-84-193, Hanscom AFB, Massabhusetts 01731.

APPENDIX A
THE PROGRAM

```
{$INCLUDE:'B:LOCGLBLS.DOC'}
{$INCLUDE:'B:LOCTMCAL.DOC'}
{$INCLUDE:'B:LOCPARMI.DOC'}
Program Simulate_locking(Input, Output, DetailFile, SummaryFile);
      {*****************************************************************}
      {* PROGRAM TO SIMULATE VARIOUS LOCKING PROTOCOLS AND TO  DETERMINE *}
      {* THEIR EFFECTS ON EXECUTION DELAY IN A DATABASE SYSTEM.          *}
      {*                                                                 *}
      {* WRITTEN BY Donald A. Varvel, August, 1984                       *}
      {* This program was written as part of the author's work on an     *}
      {* USAF-SCEEE grant June, 1984-August, 1984, and is based on ideas *}
      {* developed by Donald A. Varvel and William Perrizo.              *}
      {*                                                                 *}
      {* The files Input and Output are assumed to be interactive and    *}
      {* are used to obtain parameters.  DetailFile receives a record of *}
      {* each lock attempt and the processing of each lockable unit.  It *}
      {* should usually be NUL.  SummaryFile is the main simulation      *}
      {* output, and should be CON (CRT screen) or PRN (the printer).    *}
      {*                                                                 *}
      {* This version of this program is written for Microsoft Pascal    *}
      {* running under MS-DOS on a Zenith Z-100.  Because of limitations *}
      {* on disk space at compile time, it has been divided into a       *}
      {* PROGRAM and several UNITs and MODULEs.                          *}
      {*                                                                 *}
      {* OPERATION:                                                      *}
      {* The main program sets some parameters and then generates a      *}
      {* number of simulated transactions on which to try the various    *}
      {* combinations of transaction managers and lock managers.         *}
      {* MAKETRANS generates transactions randomly within the given      *}
      {* parameters and MAKEDELAYS generates random processing delays.   *}
      {* These are the only random processes in the simulation.  All of *}
      {* the transaction managers are run against the same transactions  *}
      {* (by SIMTRANS), so the results for a given transaction are       *}
      {* strictly comparable.                                            *}
      {*                                                                 *}
      {* As a standard of comparison, the procedure OPTIMAL has been     *}
      {* provided.  It operates with more information than a real        *}
      {* transaction manager would have, and so does not represent a     *}
      {* practical implementation.                                       *}
      {*****************************************************************}

Uses Globids, Tmcal, Parmi;   { Unit interfaces }

Var
   I, J : Integer;                  { Loop control, etc. }
   ID : 1..Algos;                   { Used in generating totals }
   Norm1, Norm2 : Real;             { Normally-distributed random numbers }
   Seed : Integer4;                 { Uniform random number seed }
   P : T_L_Ptr;                     { Auxiliary pointer for Units }
   InStr : Lstring(25);             { Input string for overriding defaults }
   Answer : Char;                    { Single-character input }
   N_To_Sim : Integer;               { Number of transactions to simulate }

{*******************************************************************}
Procedure Optimal(N_Units : Integer); Extern;

{*******************************************************************}
Function Max(A, B : Integer) : Integer; Extern;

{*******************************************************************}
Function RANDOM(var Seed : Integer4) : Real; Extern;

{*******************************************************************}
Procedure NORMAL(Var Seed : Integer4; Var Result1, Result2 : Real); Extern;

{*******************************************************************}
Procedure TM0(LU_Num : Integer);
         { Simulates the usual blocking transaction }
         { manager with calls to lock manager 0.    }
Var
   I : Integer;

Begin
   Present_time := 0;                               { Simulated clock }
   For I := 1 to LU_Num do begin
      Avail[I] := Maxint;                           { Initialization }
      Present_time := Present_time + LM0(I);        { Get lock }
      Writeln(DetailFile, 'Lock #', I:3, '       ', Present_time:6);
      Present_time := Present_time + Delay[I];      { Process }
      Writeln(DetailFile, 'Process #', I:3, '    ', Present_time:6)
   End;
   Accumulate(Present_time, LU_Num, ID);            { For averages }
   Summary_Stats(FLOAT(Present_time), FLOAT(LU_Num))
End;

{*******************************************************************}
Procedure TM1(Function Lock_Man(Loc_Num: Integer) : Boolean; LU_Num : Integer);
         { Transaction manager 1 from the paper:        }
         { While unprocessed units remain do begin      }
         {    Repeat Make nonblocking pass through list }
         {    Until a pass acquires no locks;           }
         {    Issue a blocking request for some unit    }
         { End                                          }
Var
   I, LM_Calls : Integer;
   Flag : Boolean;
   Done : Boolarray;
   Remaining : Integer;

Begin { TM1 }
                  { INITIALIZATIONS }
   ID := ID + 1;
   Remaining := LU_Num;
   LM_Calls := 0;
```

```
  Present_time := 0;
  For I := 1 to LU_Num do begin
     Done[I] := False;
     Avail[I] := Maxint
  End;

                 { NON-BLOCKING PASSES }
  While Remaining > 0 do begin
     Repeat
        Flag := False;          { Flag records recent lock acquisition }
        For I := 1 to LU_Num do
           If Not Done[I] then begin
              LM_Calls := LM_Calls + 1;
              If Lock_Man(I) then begin
                 Present_time := Present_time + Lock_Request_Delay;
                 Flag := True;
                 Writeln(DetailFile, 'Lock #', I:3, '     ', Present_time:6);
                 Present_time := Present_time + Delay[I];
                 Writeln(DetailFile, 'Process #', I:3, ' ', Present_time:6);
                 Remaining := Remaining - 1;
                 Done[I] := True
              End      { Then part }
              Else begin
                 Present_time := Present_time + Lock_Request_Delay;
                 Writeln(DetailFile, 'Lock #', I:3, '(u) ', Present_time:6)
              End
           End      { If Not Done[I] ... }
     Until Not Flag;

                        { WAIT FOR A LOCKABLE UNIT }
                        { Find first not done }
     I := 1;
     While Done[I] and (I < LU_Num) do I := I + 1;
                        { Blocking lock request }
     If Not Done[I] then begin
        Present_time := Present_time + LMO(I);
        LM_Calls := LM_Calls + 1;
        Remaining := Remaining - 1;
        Done[I] := True;
        Writeln(DetailFile, 'Lock #', I:3, '(w) ', Present_time:6);
        Present_time := Present_time + Delay[I];
        Writeln(DetailFile, 'Process #', I:3, ' ', Present_time:6)
     End      { If Not Done[I] e.g., the blocking lock request }
  End;     { While Remaining > 0 ... }

                        { Summary stats }
  Accumulate(Present_time, LM_Calls, ID);
  Summary_stats(FLOAT(Present_time), FLOAT(LM_Calls))
End;

{****************************************************************}
Procedure TM2(Function Lock_Man(Loc_Num: Integer): Boolean; LU_Num : Integer);
         { Transaction manager 2 from the paper:    }
         { While unprocessed units remain do begin }
         {    Make a nonblocking pass;             }
         {    If unprocessed units remain then     }
         {       Issue blocking request for a unit  }
         { End                                      }
Var
   I, LM_Calls : Integer;
   Done : Boolarray;
   Remaining : Integer;

Begin
               { INITIALIZATIONS }
   ID := ID + 1;
   Remaining := LU_Num;
   LM_Calls := 0;
   Present_time := 0;
   For I := 1 to LU_Num do begin
      Done[I] := False;
      Avail[I] := Maxint
   End;

   While Remaining > 0 do begin
                  { Nonblocking pass }
      For I := 1 to LU_Num do
         If Not Done[I] then begin
            LM_Calls := LM_Calls + 1;
            If Lock_Man(I) then begin
               Present_time := Present_time + Lock_Request_Delay;
               Writeln(DetailFile, 'Lock #', I:3, '     ', Present_time:6);
               Present_time := Present_time + Delay[I];
               Writeln(DetailFile, 'Process #', I:3, ' ', Present_time:6);
               Remaining := Remaining - 1;
               Done[I] := True
            End      { Then part }
            Else begin
               Present_time := Present_time + Lock_Request_Delay;
               Writeln(DetailFile, 'Lock #', I:3, '(u) ', Present_time:6)
            End
         End;     { If Not Done[I] ... }

                  { WAIT FOR A LOCKABLE UNIT }
                  { Find first not done }
      I := 1;
      While Done[I] and (I < LU_Num) do I := I + 1;
      If Not Done[I] then begin               { Blocking lock request }
         Present_time := Present_time + LMO(I);
         LM_Calls := LM_Calls + 1;
         Remaining := Remaining - 1;
         Done[I] := True;
         Writeln(DetailFile, 'Lock #', I:3, '(w) ', Present_time:6);
         Present_time := Present_time + Delay[I];
         Writeln(DetailFile, 'Process #', I:3, ' ', Present_time:6)
      End      { The blocking lock request }
   End;     { While Remaining > 0 ... }
```

```
                        { Generate summary totals }
   Accumulate(Present_time, LM_Calls, ID);
   Summary_stats(FLOAT(Present_time), FLOAT(LM_Calls))
End;

{*********************************************************************}
Procedure TM3(Function Lock_Man(Loc_Num: Integer): Boolean; LU_Num: Integer);
         { Transaction manager 3 from the paper:        }
         {    Process list using nonblocking requests; }
         {    Process list using blocking requests      }
Var
   I, LM_Calls : Integer;
   Done : Boolarray;

Begin
            { INITIALIZATIONS }
   ID := ID + 1;
   LM_Calls := 0;
   Present_time := 0;
   For I := 1 to LU_Num do begin
      Done[I] := False;
      Avail[I] := Maxint
   End;

            { NON-BLOCKING PASS }
   For I := 1 to LU_Num do begin
      LM_Calls := LM_Calls + 1;
      If Lock_Man(I) then begin
         Present_time := Present_time + Lock_Request_Delay;
         Writeln(DetailFile, 'Lock #', I:3, '     ', Present_time:6);
         Present_time := Present_time + Delay[I];
         Writeln(DetailFile, 'Process #', I:3, ' ', Present_time:6);
         Done[I] := True
      End      { Then part }
      Else begin
         Present_time := Present_time + Lock_Request_Delay;
         Writeln(DetailFile, 'Lock #', I:3, '(u) ', Present_time:6)
      End
   End;     { For }

            { BLOCKING PASS }
   For I := 1 to LU_Num do
      If Not Done[I] then begin
         LM_Calls := LM_Calls + 1;
         Present_time := Present_time + LMO(I);
         Writeln(DetailFile, 'Lock #', I:3, '(w) ', Present_time:6);
         Present_time := Present_time + Delay[I];
         Writeln(DetailFile, 'Process #', I:3, ' ', Present_time:6)
      End;     { If and For }
            { Summary }
   Accumulate(Present_time, LM_Calls, ID);
   Summary_stats(FLOAT(Present_time), FLOAT(LM_Calls))
End;
```

```
{*****************************************************************}
Procedure TM4(LU_Num : Unit_Range); Extern;

{*****************************************************************}
Procedure Getanswer(Consts S: String; Var Answer: Char); Extern;
               { Get a 1-character response from keyboard }

{*****************************************************************}
Procedure Add_Links(Var List : T_L_Ptr; Start, Finish : Integer); Extern;

{*****************************************************************}
Procedure Terminate(Var List : T_L_Ptr); Extern;

{*****************************************************************}
Procedure MakeTrans(Var U : Un_Vec);
Var
   I, Tick, Lim, Duration : Integer;
Begin
   For I := 1 to LU_No do begin
            { Queue? }
      If RANDOM(Seed) <= Q_Prob then begin
         New(U[I]);
         NORMAL(Seed, Norm1, Norm2);
         Duration := TRUNC(FLOAT(Q_Std_Dev) * Norm1) + Q_Mean_Len;
         If Duration > 0 then begin
            New(U[I]);
            U[I]^.Time := Duration;
            U[I]^.Next := Nil
         End      { Then }
      End      { Then }
            { No queue; simulate activity }
      Else begin
         U[I] := Nil;
         Tick := (-Lock_Bar - 2 * Lock_Sigma) DIV 10 * 10;
         Lim := -4 * Tick;
         While Tick < Lim do
            If RANDOM(Seed) > Ad_Req then Tick := Tick + 10
            Else begin
               NORMAL(Seed, Norm1, Norm2);
               Duration := TRUNC(FLOAT(Lock_Sigma) * Norm1) + Lock_Bar;
               If Duration <= 0 then Tick := Tick + 10
               Else begin
                  Add_Links(U[I], Tick, Duration + Tick);
                  Tick := (Tick + Duration) DIV 10 * 10 + 10
               End      { Else }
            End;     { Else and While }
         If U[I] <> Nil then Terminate(U[I])
      End      { Else }
   End      { For }
End;

{*****************************************************************}
Procedure DispTrans;
                        { Display transaction }
```

```
Var P : T_L_Ptr;
    I : Integer;
Begin
   Writeln(SummaryFile); Writeln(SummaryFile);
   Writeln(SummaryFile, 'Transaction (* = steady-state queue of given length)');
   Writeln(SummaryFile, 'Unit   Delay    Activity    Activity       ...');
   For I := 1 to LU_No do begin
      Write(SummaryFile, I:4, '   ', Delay[I]:3, '     ');
      P := Units[I];
      If P <> Nil then begin
         If P^.Next = Nil then Write(SummaryFile, '*', P^.Time:8)
         Else Repeat
            Write(SummaryFile, P^.Time:4, '-', P^.Next^.Time:4, '   ');
            P := P^.Next^.Next
         Until P^.Next = Nil
      End;
      Writeln(SummaryFile)
   End
End;

{*****************************************************************}
Procedure SimTrans;
                        { Simulate }
Var I : Integer;
Begin
   Writeln(SummaryFile);
   Write(SummaryFile,' TM    LM    Time Active  Lock Requests');
   Writeln(SummaryFile, '   Evaluation  Eval/OptEval');

   Write(SummaryFile, '   Optimal ');
   Optimal(LU_No);

   Write(SummaryFile, '   0     0 ');
   ID := 1;
   TM0(LU_No);

   Write(SummaryFile, '   1     1 ');
   TM1(LM1, LU_No);

   Write(SummaryFile, '   1     2 ');
   TM1(LM2, LU_No);

   Write(SummaryFile, '   2     1 ');
   TM2(LM1, LU_No);

   Write(SummaryFile, '   2     2 ');
   TM2(LM2, LU_No);

   Write(SummaryFile, '   3     1 ');
   TM3(LM1, LU_No);

   Write(SummaryFile, '   3     2 ');
   TM3(LM2, LU_No);
   Write(SummaryFile, '   4      3 ');
   TM4(LU_No)
End;

{*********************************************************************}
Procedure MakeDelays(Var D : Intarray; LNo : Integer;
                     Var S : Integer4); Extern;
               { Generate random processing delays }
               { uniformly from 3 to 15.           }


{*********************************************************************}
Begin                           { MAIN PROGRAM }
   Rewrite(DetailFile);
   Rewrite(SummaryFile);
   Lock_Request_Delay := 1;

   Getanswer('Do you wish to override default setup values (Y/N)?', Answer);
   De_fault;
   While ( Answer = 'Y') or (Answer = 'y') do begin
      Override;
      Display(Output);
      Getanswer('Any more changes (Y/N)? ', Answer)
   End;
   Display(SummaryFile);

   Write('Random number seed? ');
   Readln(Seed);
   Seed := Seed MOD 32768;       { Avoid overflow on first call to RANDOM }

   Repeat                        { Main program loop }

      For I := 0 to Algos do begin  { Clear totals }
         Totals[I].Time := 0;
         Totals[I].Requests := 0
      End;

      Repeat                     { Minor input loop }
         Write('Enter number of lockable units per transaction: ');
         Readln(LU_No);
         If (LU_No < 1) or (LU_No > Max_Units) then
            Writeln('Must be in 0 < N <= ', Max_Units:1, '.')
      Until (LU_No > 0) and (LU_No <= Max_Units);

      Repeat                     { Minor input loop }
         Write('Enter number of transactions to simulate: ');
         Readln(N_To_Sim);
         If (N_To_Sim < 1) then Writeln('Must be positive.')
      Until (N_To_Sim > 0);
                                 { Simulate N_To_Sim transactions }
      For I := 1 to N_To_Sim do begin
         MakeTrans(Units);
         MakeDelays(Delay, LU_No, Seed);
         DispTrans;
         SimTrans
      End;
      Averages(N_To_Sim);
      Getanswer('Simulate another transaction? (Y/N) ', Answer)
   Until (Answer <> 'Y') and (Answer <> 'y')
End.


{$INCLUDE:'B:LOCGLBLS.DOC'}
{$INCLUDE:'B:LOCTMCAL.DOC'}

      {****************************************************************}
      {* Module containing code to simulate the imaginary algorithm *}
      {* OPTIMAL and the 4th modified transaction manager.  Each      *}
      {* contains some internal procedures; notably, in the case of *}
      {* TM4, the two parts of the third modified lock manager,      *}
      {* LM3I and LM3W.                                               *}
      {****************************************************************}
Module Locopt;
Uses Globids, Tmcal;

Function Max(A, B : Integer) : Integer; Extern;

Procedure Optimal(LU_Num : Integer);
      {****************************************************************}
      {* OPTIMAL determines a lower bound on processing the given    *}
      {* transaction using locks.  By assumption it uses only as     *}
      {* many lock requests as there are lockable units and gets     *}
      {* into all queues at initiation time.  It does a search of    *}
      {* the decision tree of orders of lock requests to find one    *}
      {* that results in the least delay.  The treesearch selects    *}
      {* a first order of requests that is likely to be good, and    *}
      {* performs forward pruning according to two criteria; its     *}
      {* worst-case performance is O(N!), but is usually O(N).       *}
      {*                                                              *}
      {* OPTIMAL contains the recursive treesearch Findbest, which  *}
      {* in turn contains Sort.                                       *}
      {****************************************************************}
Type
   Low_Rec = Record
      When : Integer;
      Proc : 3..15
   End;

Var
   Cutoff : Integer;              { Best time found so far.  If it can't   }
                                  { be undercut, prune the present branch. }
   I : Integer;
   Remaining : Unit_Range;        { How many units remain to be processed? }
   Done : Boolarray;
   Low_Vec : Array[0..Max_Units] of Low_Rec;
                                  { Used in computing Lowerbound }
```

```
Function Findbest : Integer;
     (**************************************************)
     (*  Recursive decision tree search, with cutoffs. *)
     (**************************************************)
Type
   ND_Rec = Record              { Units not yet done and weights for sorting }
      U : Unit_Range;
      Val : Integer
   End;
   ND_Vec = Array[Unit_Range] of ND_Rec;

Var
   Getlock, Best_Path, Lowerbound, Pathtime : Integer;
   Cursor : 0..Max_Units;
   I : Unit_Range;
   Notdone : ND_Vec;
   P : T_L_Ptr;
   J, Cum_Delay : Integer;

Procedure Sort(Var Tosort : ND_Vec; N : Unit_Range);
               {*******************************************************}
               {* Linear insertion sort, in place.  An O(N**2) sort *}
               {* makes sense here, since it will be called far     *}
               {* more times with small N than with large.  This    *}
               {* sort beats Shellsort and Quicksort for N less     *}
               {* than about 15, and N will seldom be that large.   *}
               {*******************************************************}
Var
   I, J, TempVal : Integer;
   TempU : Unit_Range;
Begin
   For I := 1 to N-1 do begin     { Elements 1..I are in order }
      TempU := Tosort[I+1].U;
      TempVal := Tosort[I+1].Val;
      J := I;
      While TempVal < Tosort[J].Val do begin
         Tosort[J+1].U := Tosort[J].U;
         Tosort[J+1].Val := Tosort[J].Val;
         J := J - 1;
         If J < 1 then Break     { Nonstandard Pascal: Leave innermost loop }
      End;     { While }
      Tosort[J+1].U := TempU;
      Tosort[J+1].Val := TempVal
   End      { For I ... }
End;

{*************************************************}
Begin    { Findbest }
   If Remaining = 1 then begin
            { Only one unit remains }
      I := 1; While Done[I] do I := I + 1;
      Getlock := LMO(I) + Delay[I];
      If Present_time + Getlock < Cutoff then Cutoff := Present_time + Getlock;
      Findbest := Getlock
```

```
End      { Else if Remaining = 1 ... }
Else begin
         { More than one unit remains }
   Best_Path := Maxint;
   Lowerbound := 0;
   Cursor := 0;
         { Compute Lowerbound }
   For I := 1 to LU_Num do
      If Not Done[I] then begin
         Cum_Delay := LMO(I);
         Low_Vec[0].When := Cum_Delay;
         Low_Vec[0].Proc := Delay[I];
         J := Cursor;
         While Low_Vec[J].When < Cum_Delay do begin
            Low_Vec[J+1] := Low_Vec[J];
            J := J - 1
         End;     { While }
         Cursor := Cursor + 1;
         Low_Vec[J+1] := Low_Vec[0]
      End;     { Then and For }
   Cum_Delay := 0;
   For I := 1 to Cursor do begin
      Cum_Delay := Cum_Delay + Low_Vec[I].Proc;
      Lowerbound := Max(Lowerbound, Low_Vec[I].When + Cum_Delay);
      Cum_Delay := Cum_Delay + Lock_Request_Delay
   End;

   If Present_time + Lowerbound < Cutoff then begin
         { Arrange those units not processed }
         { Generate weights: Time of next locking for those units  }
         {    that are unlocked but which will become locked again }
         {    (Note use of crystal ball), Processing-time + 9000   }
         {    for those that are available and will not become     }
         {    unavailable, and Release-time + 10000 for those that }
         {    are presently locked.                                }
      Cursor := 0;
      For I := 1 to LU_Num do
         If Not Done[I] then begin
            Cursor := Cursor + 1;
            Notdone[Cursor].U := I;
            If Units[I] = Nil then Notdone[Cursor].Val := Delay[I] + 9000
            Else if(Units[I]^.Next = Nil)and(Units[I]^.Time <= Present_time)
               then Notdone[Cursor].Val := Delay[I] + 9000
            Else if Units[I]^.Next = Nil then
               Notdone[Cursor].Val := Units[I]^.Time + 10000
            Else begin
               P := Units[I];
               While (P^.Next^.Time <= Present_time)
                     and (P^.Next^.Next^.Next <> Nil)
                  do P := P^.Next^.Next;
                                 {  PT  ()  }
               If P^.Time > Present_time then Notdone[Cursor].Val := P^.Time
                                 {  ( PT )  }
               Else if P^.Next^.Time > Present_time then
```

```
                    Notdone[Cursor].Val := P^.Next^.Time + 10000
                                         {  ()  PT  }
                 Else Notdone[Cursor].Val := Delay[I] + 9000
              End      { Else }
           End;     { Then }

              { Sort according to weights }
        Sort(Notdone, Cursor);

              { Search for optimal order, cutting off if equal to }
              { a previously-computed lower bound or if unable to }
              { better the best previous time.                    }
        I := 1;
        While (I <= Cursor) and (Best_Path > Lowerbound) do begin
           Getlock := LMO(Notdone[I].U) + Delay[Notdone[I].U];
                    { Simulate processing the unit }
           Done[Notdone[I].U] := True;
           Present_time := Present_time + Getlock;
           Remaining := Remaining - 1;
                    { Recurse }
           Pathtime := Findbest;
                    { Record best found so far }
           If Pathtime < Best_Path - Getlock then
              Best_Path := Pathtime + Getlock;
                    { Undo }
           Done[Notdone[I].U] := False;
           Present_time := Present_time - Getlock;
           Remaining := Remaining + 1;
              { Increment loop control }
           I := I + 1
        End     { While }
     End;     { Then }
     Findbest := Best_Path
  End      { Else }
End;     { Findbest }

{***********************************************************}
Begin    { Optimal }
              { Initializations }
  Remaining := LU_Num;
  Cutoff := Maxint;
  Present_Time := 0;
  For I := 1 to LU_Num do begin
     Done[I] := False;
     Avail[I] := Maxint;
             { Start all queues }
     If Units[I] <> Nil then if Units[I]^.Next = Nil then
        Avail[I] := Units[I]^.Time
  End;

              { Call recursive treesearch }
  I := Findbest;

  Opteval(FLOAT(Cutoff), FLOAT(LU_Num));       { Record, for comparison }
```

```
   Accumulate(Cutoff, LU_Num, 0);
   Summary_Stats(FLOAT(Cutoff), FLOAT(LU_Num))
End;     { Optimal }

{*****************************************************************}
{*****************************************************************}
Procedure TM4(LU_Num : Unit_Range);
               { Transaction manager 4:                 }
               {    Make a nonblocking pass through list; }
               {    While units remain do begin          }
               {       Wait for one to become available;  }
               {       Process it                         }
               {    End                                   }
Type
   OR_Ptr = ^OR_Rec;
   OR_Rec = Record
      LU : Unit_Range;
      Next : OR_Ptr
   End;

Var
   OR_List : OR_Ptr;
   LM_Calls, Remaining, I : Integer;
   Selected : Unit_Range;

{*****************************************************************}
Procedure AddtoOR(LU : Unit_Range; Var List : OR_Ptr);
               { Add a lockable unit to this transaction's wait-list }
Var P : OR_Ptr;
Begin
   New(P);
   P^.Next := List;
   P^.LU := LU;
   List := P
End;

{*****************************************************************}
Procedure DelOR(Var List : OR_Ptr; Val : Unit_Range);
               { Delete a lockable unit from this transaction's wait-list }
Var P : OR_Ptr;
Begin
   If List <> Nil then begin
      If List^.LU = Val then begin
         P := List;
         List := List^.Next;
         Dispose(P)
      End
      Else DelOR(List^.Next, Val)
   End
End;

{*****************************************************************}
Function LM3I(LU : Unit_Range) : Boolean;
               { Lock Manager 3, non-blocking part:        }
```

```
                {    If unit is available then return true }
                {    Else begin                            }
                {       Add unit to wait-list;            }
                {       Return false                      }
                {    End                                  }
Var
   P : T_L_Ptr;
   Result : Boolean;
Begin
   If Units[LU] = Nil then Result := True
   Else if Units[LU]^.Next = Nil then begin
      Result := False;
      Avail[LU] := Present_time + Units[LU]^.Time
   End
   Else if Units[LU]^.Time > Present_time then Result := True
   Else begin
      P := Units[LU];
      Result := True;
      While P^.Time <= Present_time do begin
         If P^.Next^.Time > Present_time then begin
            Result := False;
            Avail[LU] := P^.Next^.Time
         End;
         P := P^.Next^.Next
      End      { While }
   End;     { Else }
   If Not Result then AddtoOR(LU, OR_List);
   LM3I := Result
End;

{******************************************************************}
Function LM3W : Integer;
                { Lock Manager 3, blocking part:            }
                {    Set Selected to next unit available;  }
                {    Return delay for next unit            }
Var
   P : OR_Ptr;
   UP : T_L_Ptr;
   Least : Unit_Range;
   LeastVal, Waittime : Integer;

Begin
   If OR_List = Nil then LM3W := -1      { Nothing left }
   Else begin
                { Finding next is not trivial! }
      Repeat
         LeastVal := Maxint;
         P := OR_List;
         While P <> Nil do begin
            If Avail[P^.LU] < LeastVal then begin
               LeastVal := Avail[P^.LU];
               Least := P^.LU
            End;
            P := P^.Next
```

```
        End;
        If LeastVal < Present_time then
           If Units[Least]^.Next <> Nil then begin
              UP := Units[Least];
              While UP^.Time < Present_time do begin
                 If UP^.Next^.Time > Present_time then
                    Avail[Least] := UP^.Next^.Time;
                 UP := UP^.Next^.Next
              End      { While }
           End      { Ifs }
     Until Avail[Least] = LeastVal;
     DelOR(OR_List, Least);
     Selected := Least;
     LM3W := Max(Lock_Request_Delay, LeastVal - Present_time)
  End      { Else }
End;

{******************************************************************}
Begin        { Transaction Manager 4 itself }
        { Initializations }
  Remaining := 0;
  OR_List := Nil;
  LM_Calls := 0;
  Present_time := 0;

        { Non-blocking run }
  For I := 1 to LU_Num do begin
     Avail[I] := Maxint;
     LM_Calls := LM_Calls + 1;
     If LM3I(I) then begin
        Present_time := Present_time + Lock_Request_Delay;
        Writeln(Detailfile, 'Lock #', I:3, '    ', Present_time:6);
        Present_time := Present_time + Delay[I];
        Writeln(Detailfile, 'Process #', I:3, ' ', Present_time:6)
     End      { Then part }
     Else begin
        Remaining := Remaining + 1;
        Present_time := Present_time + Lock_Request_Delay;
        Writeln(Detailfile, 'Lock #', I:3, '(u) ', Present_time:6)
     End
  End;     { Non-blocking run }

        { Iteratively, wait for next available lock }
  While Remaining > 0 do begin
     Present_time := Present_time + LM3W;
     Writeln(Detailfile, 'Lock #', Selected:3, '(o) ', Present_time:6);
     Present_time := Present_time + Delay[Selected];
     Writeln(Detailfile, 'Process #', Selected:3, ' ', Present_time:6);
     Remaining := Remaining - 1;
     LM_Calls := LM_Calls + 1
  End;     { While = Wait for locks }

  Accumulate(Present_time, LM_Calls, 0);
  Summary_Stats(FLOAT(Present_time), FLOAT(LM_Calls))
```

```
End;

End.     { Module Locopt }

   {*************************************************************}
   {*  Interface that supplies global identifiers to all those *}
   {*  program units that need them.  Important consts and     *}
   {*  types are included, but also the files, lockable unit   *}
   {*  information Units, availability, processing delay, and  *}
   {*  the scalars Lock_Request_Delay (presently 1), number of *}
   {*  lockable units LU_No, and Present_time.                 *}
   {*************************************************************}
Interface;
Unit Globids(Max_Units, Algos, Unit_Range, T_L_Ptr, Time_list,
   Un_Vec, Boolarray, Intarray, Detailfile, Summaryfile, Units,
   Lock_Request_delay, Present_time, Avail, Delay, LU_No);
Const
   Max_Units = 100;                  { Max lockable units (arbitrary)}
   Algos = 8;                        { Number of algorithms         }
Type
   Unit_Range = 1..Max_Units;
   T_L_Ptr = ^Time_list;             { Time node for linked list }
   Time_list = Record
      Time : Integer;
      Next : T_L_Ptr
   End;
   Un_Vec = Array[Unit_Range] of T_L_Ptr;
                                     { Array of time lists }
   Boolarray = Array[Unit_Range] of Boolean;
                                     { Type for Done, used in subprograms }
   Intarray = Array[Unit_Range] of Integer;
                                     { Used for Avail and Delay }
Var
   DetailFile, SummaryFile : Text;   { Output files }
   Units : Un_Vec;                   { Availability of lockable units:
      +-----------+-----------+-----------+-----------+-----------+
      |     .     |     .     |     .     |     .     |     .     |
      +-----|-----+-----|-----+-----|-----+-----|-----+-----|-----+
          -----    +----V+---+    -----    +----V+---+ +----V+---+
            -      | 110 | . |      -      |   0 | . | |  20 | . |
            |      +-----+-|-+      |      +-----+-|-+ +-----+-|-+
            |            -----      |      +-----+-V-+ +-----+-V-+
         Always         |  -     Always    | 180 | . | | 140 | . |
       available        |      available   +-----+-|-+ +-----+-|-+
                    Queue of               +-----+-V-+ +-----+-V-+
                   length 110              | 300 + . | |Mxint| . |
                                           +-----+-|-+ +-----+-|-+
                                           +-----+-V-+       -----
                                           | 450 | . |     |   -
               Locked from 0 to 180        +-----+-|-+     |
                        and          >     +-----+-V-+   Locked
                    300 to 450             |Mxint| . |    from
                                           +-----+-|-+  20 to 140
```

```
                                                             -----
                                                              -         }
   Lock_Request_Delay : Integer;      { Time needed to access lock manager }
   Present_time : Integer;            { Simulated clock }
   Avail, Delay : Intarray;           { When available, Processing delay }
   LU_No : Integer;

Begin
End;


{$INCLUDE:'B:LOCGLBLS.DOC'}
Implementation of globids;
Begin
End.


{$INCLUDE:'B:LOCGLBLS.DOC'}
Module Helps;
   {*******************************************************}
   {*  This module contains certain small, relatively pure  *}
   {*  subprograms.  They are declared EXTERN by those      *}
   {*  calling units that access them.                      *}
   {*******************************************************}

Uses Globids;

{***************************************************************}
Function Max(A, B : Integer) : Integer;
         { Returns value of maximum of two args     }
Begin
   If A > B then Max := A
   Else Max := B
End;

{***************************************************************}
Function RANDOM(Var Seed : Integer4) : Real;
         { Generates uniform random floating-point numbers in the }
         { range 0 <= R < 1, using the linear congruence method   }
         { of Knuth vol. 2.  C is zero, so the low-order bits are }
         { not very random.  Test high-order bits instead.        }
Const Multiplier = 25997;
      Modulus = 32768;
      Fmod = 32768.0;
Begin
   Seed := (Seed * Multiplier) mod Modulus;
   RANDOM := FLOAT4(Seed) / Fmod
End;

{***************************************************************}
Procedure NORMAL(Var Seed : Integer4; Var Result1, Result2 : Real);
         { Generates two normally-distributed random numbers with }
         { a mean of zero and a standard deviation of one, using  }
         { a method given in Knuth vol. 2.                        }
```

```
Var
   Root, V1, V2, S : Real;
Begin
   Repeat
      V1 := 2.0 * RANDOM(Seed) - 1;
      V2 := 2.0 * RANDOM(Seed) - 1;
      S := SQR(V1) + SQR(V2)
   Until (S < 1) and (S > 0);
   Root := SQRT(-2.0 * LN(S) / S);
   Result1 := V1 * Root;
   Result2 := V2 * Root
End;

{*****************************************************************}
Procedure Getanswer(Consts S: String; Var Answer : Char);
               { Get a 1-character response from keyboard }
Begin
   Write(S);
   Readln(Answer)
End;

{*****************************************************************}
Procedure Add_Links(Var List : T_L_Ptr; Start, Finish : Integer);
               { Adds an activity entry to Units }
Begin
   If(Finish > 0) and (List = Nil) then begin
      New(List);
      List^.Time := Start;
      New(List^.Next);
      List^.Next^.Time := Finish;
      List^.Next^.Next := Nil
   End
   Else if Finish > 0 then Add_Links(List^.Next^.Next, Start, Finish)
End;

{*****************************************************************}
Procedure Terminate(Var List : T_L_Ptr);
               { Terminates activity list with (Maxint, Nil) }
Begin
   If List = Nil then begin
      New(List);
      List^.Time := Maxint;
      List^.Next := Nil
   End
   Else Terminate(List^.Next^.Next)
End;

{*****************************************************************}
Procedure MakeDelays(Var D : Interray; LNo : Integer; Var Seed : Integer4);
               { Generate random processing delays }
               { uniformly from 3 to 15.           }
Var I : Integer;
Begin
   For I := 1 to LNo do D[I] := TRUNC(13.0 * RANDOM(Seed)) + 3
```

```
End;


End.

   {************************************************}
   {*  Simulation parameter initialization routines.  *}
   {************************************************}
Interface;
Unit Parmi(Override, De_fault, Display, Ad_Req, Lock_Bar, Lock_Sigma,
           Q_Prob, Q_Mean_Len, Q_Std_Dev);

Var
   Ad_Req : Real;                  { Probability of an adverse request on }
                                   { a lockable unit in a 10-tick period  }
   Lock_Bar : Integer;             { Mean adverse lock duration }
   Lock_Sigma : Integer;           { Std. deviation of adverse lock duration }
   Q_Prob : Real;                  { Probability of a queue on a unit }
   Q_Mean_Len : Integer;           { Mean queue length }
   Q_Std_Dev : Integer;            { Std. deviation of queue length }

Procedure Override;  { Override the default on one or more parameters }
Procedure De_fault;  { Set parameters to default }
Procedure Display(Var F: Text);  { Display parameters }

End;


{$INCLUDE:'B:LOCPARMI.DOC'}

   {*******************************************************}
   {*  Implementation of parameter init & display routines.  *}
   {*******************************************************}
Implementation of Parmi;

{*******************************************************}
{*  Constants used in the initialization of simulation  *}
{*  parameters.  These are the defaults.                *}
{*******************************************************}
Const
   Con_Def = 0.01;                 { Default probability of        }
                                   { adverse lock request/10 ticks}
   Lock_Default = 150;             { Default mean lock duration    }
   Lk_Sig_Default = 50;            { Default standard deviation    }
                                   { of lock duration              }

   Q_Prob_Default = 0.1;           { Default probability of queue }
   Q_Bar = 100;                    { Default mean queue length     }
   Q_Sigma = 20;                   { Default queue std. dev.       }

{****************************************************************}
Procedure Override;  { Override the default on one or more parameters }
Var
```

```
   InStr : Lstring(25);

Begin
   Writeln('Enter number (or <ENTER> for default).');
   Write('Prob. of adverse request per 10 ticks (', Ad_Req:6:4, ')');
   Readln(InStr);
   If (InStr<>Null)then if Not DECODE(InStr, Ad_Req) then Ad_Req := Con_Def;
   Write('Mean lock duration (', Lock_Bar:1, ')');
   Readln(InStr);
   If (InStr <> Null) then if Not DECODE(InStr, Lock_Bar) then
      Lock_Bar := Lock_Default;
   Write('Std. deviation of lock duration (', Lock_Sigma:1, ')');
   Readln(InStr);
   If (InStr <> Null) then if Not DECODE(InStr, Lock_Sigma) then
      Lock_Sigma := Lk_Sig_Default;
   Write('Per-unit probability of a queue (', Q_Prob:6:4, ')');
   Readln(InStr);
   If (InStr <> Null) then if Not DECODE(InStr, Q_Prob) then
      Q_Prob := Q_Prob_Default;
   Write('Probable length of queue in ticks (', Q_Mean_Len:1, ')');
   Readln(InStr);
   If (InStr <> Null) then if Not DECODE(InStr, Q_Mean_Len) then
      Q_Mean_Len := Q_Bar;
   Write('Std. deviation of queue length (', Q_Std_Dev:1, ')');
   Readln(InStr);
   If (InStr <> Null) then if Not DECODE(InStr, Q_Std_Dev) then
      Q_Std_Dev := Q_Sigma
End;

{*****************************************************************}
Procedure De_fault;
               { Set parameters to default }
Begin
   Ad_Req := Con_Def;
   Lock_Bar := Lock_Default;
   Lock_Sigma := Lk_Sig_Default;
   Q_Prob := Q_Prob_Default;
   Q_Mean_Len := Q_Bar;
   Q_Std_Dev := Q_Sigma
End;

{*****************************************************************}
Procedure Display;
               { Display parameters to file F }
Begin
   Writeln(F, 'Parameters          Probability   Mean   Std. Dev.');
   Writeln(F, '-----------------------------------------------------');
   Writeln(F, 'Potential conflicts', Ad_Req:10:3, Lock_Bar:10,
      Lock_Sigma:8);
   Writeln(F, 'Queues              ', Q_Prob:10:3, Q_Mean_Len:10,
      Q_Std_Dev:8);
   Writeln(F)
End;
```

```
End.

   {*************************************************************}
   {*  Interface for routines called by transaction managers  *}
   {*************************************************************}
Interface;

Unit Tmcal(TotRec, Totals, LM0, LM1, LM2, Summary_stats, Accumulate,
           Averages, Opteval);

Const Algos = 8;

Type TotRec = Record Time, Requests : Integer4 End;

Var Totals : Array[0..Algos] of TotRec;

Function LM0(LU : Integer) : Integer;
         { The usual blocking lock manager, with the }
         { addition of a check to Avail to go with   }
         { lock manager 2.                           }

Function LM1(LU : Integer) : Boolean;

Function LM2(LU : Integer) : Boolean;

Procedure Summary_Stats(Time, Requests : Real);

Procedure Accumulate(Time, Requests, ID : Integer);
               { Accumulate totals for a particular trans. type }
Procedure Averages(T_Num : Integer);

Procedure Opteval(Time, Requests : Real);

End;


{$INCLUDE:'B:LOCTMCAL.DOC'}
{$INCLUDE:'B:LOCGLBLS.DOC'}
   {*************************************************************}
   {*  Implementation of Lock Managers 0-2, Summary_stats,     *}
   {*  Accumulate, Averages, and Opteval.                      *}
   {*************************************************************}
Implementation of Tmcal;
Uses Globids;

Var BestE : Real;        { Evaluation of OPTIMAL, for comparison }

Function Max(A, B : Integer) : Integer; Extern;

Function LM0;
         { The usual blocking lock manager, with the addition of }
         { a check to Avail to go with lock manager 2.           }
Var P : T_L_Ptr;
```

```
Begin
   If Units[LU] = Nil Then LM0 := Lock_Request_Delay
   Else if Avail[LU] <> Maxint Then
      LM0 := Max(Avail[LU] - Present_time, Lock_Request_Delay)
   Else if Units[LU]^.Next = Nil Then
      LM0 := Max(Lock_Request_Delay, Units[LU]^.Time)
   Else if Units[LU]^.Time > Present_Time Then
      LM0 := Lock_Request_Delay
   Else begin
      P := Units[LU];
      LM0 := Lock_Request_Delay;
      While P^.Time <= Present_time do begin
         If P^.Next^.Time > Present_time then
            LM0 := Max(P^.Next^.Time - Present_time, Lock_Request_delay);
         P := P^.Next^.Next
      End
   End
End;

{*******************************************************************}
Function LM1;
        { A simulator of Lock Manager 1: This is the nonblocking part. }
        { LM0 is used for the blocking part.  This lock manager simply }
        { determines whether the requested lock is available and       }
        { returns true or false.                                       }
Var P : T_L_Ptr;
Begin
   If Units[LU] = Nil then LM1 := True               { No activity }
   Else if Units[LU]^.Next = Nil Then LM1 := False   { Queue }
   Else if Units[LU]^.Time > Present_time Then LM1 := True  { Not yet started }
   Else begin  { Do an activity's start and finish bracket Present_time? }
      P := Units[LU];
      LM1 := True;
      While P^.Time <= Present_time do begin
         If P^.Next^.Time > Present_time then LM1 := False;
         P := P^.Next^.Next
      End      { While P^.Time ... }
   End      { Else begin ... }
End;

{*******************************************************************}
Function LM2;
        { Simulator of Lock Manager 2:                       }
        { This is the nonblocking part only.  LM0 is used for the }
        { blocking part.  This lock manager simulates placing the }
        { transaction in the queue if the unit is not immediately }
        { available.                                              }
Var P : T_L_Ptr;
Begin
   If Units[LU] = Nil then LM2 := True    { No activity }
   Else if Avail[LU] <= Present_time then LM2 := True  { Presently available }
   Else if Avail[LU] <> Maxint then LM2 := False  { Not available yet }
   Else if Units[LU]^.Next = Nil then begin    { Queue }
      Avail[LU] := Present_time + Units[LU]^.Time;
      LM2 := False
   End
   Else if Units[LU]^.Time > Present_time then LM2 := True  { Future activity }
   Else begin   { Do an activity's start and finish bracket Present_time? }
      P := Units[LU];
      LM2 := True;
      While P^.Time <= Present_time do begin
         If P^.Next^.Time > Present_time then begin
            LM2 := False;
            Avail[LU] := P^.Next^.Time
         End;
         P := P^.Next^.Next
      End       { While P^.Time ... }
   End      { Else ... }
End;

{****************************************************************}
Procedure Summary_Stats;
Var Eval : Real;
Begin
   Write(SummaryFile, Time:11:1);
   Write(SummaryFile, Requests:12:1);
   Eval := (1.0 + SQR(1.0 - Requests/LU_No)) * Time;
   Write(SummaryFile, Eval:15:2);
   Writeln(SummaryFile, Eval / BestE :15:3)
End;

{****************************************************************}
Procedure Accumulate;
               { Accumulate totals for a particular trans. type }
Begin
   Totals[ID].Time := Totals[ID].Time + Time;
   Totals[ID].Requests := Totals[ID].Requests + Requests
End;

{****************************************************************}
Procedure Averages;
               { Display averages for a particular trans. type }
Var I : Integer;
Begin
   Opteval(Totals[0].Time / T_Num, Totals[0].Requests / T_num);
   Writeln(SummaryFile); Writeln(SummaryFile, '**************************');
   Writeln(SummaryFile, 'Averages for this transaction type:');
   Write(SummaryFile,'  TM    LM    Time Active  Lock Requests');
   Writeln(SummaryFile, '   Evaluation  Eval/OptEval');
   Write(Summaryfile, '  Optimal  ');
   Summary_stats(Totals[0].Time / T_Num, Totals[0].Requests / T_Num);
   Write(Summaryfile, '   0     0 ');
   Summary_stats(Totals[1].Time / T_Num, Totals[1].Requests / T_Num);
   For I := 2 to Algos-1 do begin
      Write(Summaryfile, I DIV 2 : 4, I MOD 2 + 1 : 6, ' ');
      Summary_stats(Totals[I].Time / T_Num, Totals[I].Requests / T_Num)
   End;
   Write(Summaryfile, 4:4, 3:6, ' ');
   Summary_stats(Totals[8].Time / T_num, Totals[8].Requests / T_Num)
End;

Procedure Opteval;
               { Record evaluation of OPTIMAL for comparison with others }
Begin
   BestE := (1.0 + SQR(1.0 - Requests / LU_No)) * Time
End;

End.
```

APPENDIX B: RESULTS

LIGHT ACTIVITY

Parameters	Probability	Mean	Std. Dev.
Potential conflicts	0.004	150	50
Queues	0.080	250	80

Averages for 5 lockable units:

TM	LM	Time Active	Lock Requests	Evaluation	Eval/OptEval
Optimal		136.1	5.0	136.05	1.000
0	0	162.4	5.0	162.45	1.194
1	1	160.5	6.2	168.99	1.242
1	2	145.1	6.2	152.72	1.123
2	1	160.1	5.6	162.41	1.194
2	2	145.1	5.6	147.14	1.082
3	1	160.1	5.6	161.99	1.191
3	2	145.1	5.6	146.81	1.079
4	3	144.4	5.6	146.15	1.074

Averages for 10 lockable units:

TM	LM	Time Active	Lock Requests	Evaluation	Eval/OptEval
Optimal		230.0	10.0	230.00	1.000
0	0	376.2	10.0	376.20	1.636
1	1	353.5	13.6	400.65	1.742
1	2	261.0	13.5	292.92	1.274
2	1	352.5	12.4	372.02	1.617
2	2	261.0	12.3	274.16	1.192
3	1	351.9	11.6	361.43	1.571
3	2	261.0	11.6	268.05	1.165
4	3	259.0	11.6	266.00	1.157

Averages for 15 lockable units:

TM	LM	Time Active	Lock Requests	Evaluation	Eval/OptEval
Optimal		244.6	15.0	244.55	1.000
0	0	476.3	15.0	476.25	1.947
1	1	452.5	20.0	501.72	2.052
1	2	300.4	19.6	329.27	1.346
2	1	450.9	18.1	470.11	1.922
2	2	300.9	18.0	312.49	1.278
3	1	449.9	17.0	457.90	1.872
3	2	300.9	17.0	306.20	1.252
4	3	298.3	17.0	303.55	1.241

Averages for 20 lockable units:

TM	LM	Time Active	Lock Requests	Evaluation	Eval/OptEval
Optimal		270.3	20.0	270.25	1.000
0	0	718.4	20.0	718.40	2.658
1	1	651.3	27.1	733.32	2.714
1	2	354.1	26.7	393.84	1.457
2	1	655.2	24.5	688.37	2.547
2	2	354.4	24.3	370.35	1.370
3	1	653.6	22.6	664.65	2.459
3	2	355.5	22.6	361.46	1.337
4	3	352.0	22.6	357.95	1.325

MEDIUM ACTIVITY

Parameters	Probability	Mean	Std. Dev.
Potential conflicts	0.010	150	50
Queues	0.100	250	80

Averages for 5 lockable units:

TM	LM	Time Active	Lock Requests	Evaluation	Eval/OptEval
Optimal		183.5	5.0	183.50	1.000
0	0	259.0	5.0	259.00	1.411
1	1	254.0	8.3	361.32	1.969
1	2	192.9	8.1	264.61	1.442
2	1	253.4	6.9	291.88	1.591
2	2	192.9	6.8	217.84	1.187
3	1	252.8	6.3	269.84	1.470
3	2	192.9	6.3	205.89	1.122
4	3	190.8	6.3	203.70	1.110

Averages for 10 lockable units:

TM	LM	Time Active	Lock Requests	Evaluation	Eval/OptEval
Optimal		207.6	10.0	207.55	1.000
0	0	427.3	10.0	427.25	2.059
1	1	382.5	15.5	498.21	2.400
1	2	243.2	15.1	305.22	1.471
2	1	381.0	13.1	418.86	2.018
2	2	244.9	12.8	263.42	1.269
3	1	381.0	12.3	401.10	1.933
3	2	245.9	12.3	258.91	1.247
4	3	236.1	12.3	248.54	1.197

Averages for 15 lockable units:

TM	LM	Time Active	Lock Requests	Evaluation	Eval/OptEval
Optimal		257.1	15.0	257.10	1.000
0	0	586.2	15.0	586.20	2.280
1	1	486.4	22.3	599.97	2.334
1	2	303.1	21.7	363.57	1.414
2	1	484.0	19.3	522.91	2.034
2	2	305.0	18.9	325.04	1.264
3	1	491.0	17.6	506.32	1.969
3	2	306.3	17.6	315.81	1.228
4	3	298.0	17.6	307.30	1.195

Averages for 20 lockable units:

TM	LM	Time Active	Lock Requests	Evaluation	Eval/OptEval
Optimal		299.0	20.0	299.00	1.000
0	0	845.0	20.0	845.05	2.826
1	1	663.0	29.7	819.02	2.739
1	2	383.8	28.4	451.50	1.510
2	1	660.8	26.3	725.39	2.426
2	2	385.5	25.5	415.13	1.388
3	1	661.2	24.0	686.99	2.298
3	2	390.4	24.0	405.58	1.356
4	3	380.5	24.0	395.29	1.322

HEAVY ACTIVITY

Parameters	Probability	Mean	Std. Dev.
Potential conflicts	0.050	150	50
Queues	0.150	250	80

Averages for 5 lockable units:

TM	LM	Time Active	Lock Requests	Evaluation	Eval/OptEval
Optimal		201.6	5.0	201.55	1.000
0	0	407.4	5.0	407.40	2.021
1	1	328.4	11.2	833.22	4.134
1	2	222.1	10.8	515.94	2.560
2	1	328.0	8.7	507.54	2.518
2	2	222.1	8.1	310.32	1.540
3	1	334.7	7.3	405.52	2.012
3	2	223.4	7.3	270.73	1.343
4	3	218.3	7.3	264.43	1.312

Averages for 10 lockable units:

TM	LM	Time Active	Lock Requests	Evaluation	Eval/OptEval
Optimal		278.1	10.0	278.10	1.000
0	0	732.5	10.0	732.45	2.634
1	1	528.0	26.2	1913.50	6.881
1	2	320.1	23.5	903.62	3.249
2	1	529.5	19.8	1037.93	3.732
2	2	320.1	18.0	527.62	1.897
3	1	601.7	15.3	767.54	2.760
3	2	326.8	15.3	416.87	1.499
4	3	307.1	15.3	391.74	1.409

Averages for 15 lockable units:

TM	LM	Time Active	Lock Requests	Evaluation	Eval/OptEval
Optimal		290.0	15.0	289.95	1.000
0	0	1097.6	15.0	1097.60	3.785
1	1	656.8	45.0	3284.00	11.326
1	2	342.3	42.6	1500.97	5.177
2	1	696.3	33.3	1732.80	5.976
2	2	352.5	30.0	702.75	2.424
3	1	798.5	23.5	1051.97	3.628
3	2	367.6	23.5	484.26	1.670
4	3	330.6	23.5	435.58	1.502

Averages for 20 lockable units:

TM	LM	Time Active	Lock Requests	Evaluation	Eval/OptEval
Optimal		328.8	20.0	328.75	1.000
0	0	1381.4	20.0	1381.40	4.202
1	1	977.3	57.3	4376.79	13.313
1	2	391.6	46.8	1097.38	3.338
2	1	980.5	41.8	2150.89	6.543
2	2	397.9	36.3	662.20	2.014
3	1	1154.2	30.4	1463.30	4.451
3	2	418.3	30.4	530.26	1.613
4	3	394.5	30.4	500.21	1.522

DISTRIBUTION LIST

HQ USAF/XOORC
Washington, D.C. 20330 (1)

Air Force Systems Command
Andrews AFB, MD 20332
AFSC/XR (1)
AFSC/XRK (2)

Electronic Systems Division
Hanscom AFB, MA 01731
AFGL/SULR (3)
AFGL/SULL (1)
ESD/CC (1)
ESD/DC (1)
ESD/IN (1)
ESD/OC (1)
ESD/TC (2)
ESD/TO (1)
ESD/YW (1)
ESD/XR (2)
ESD/XRC (2)
ESD/XRX (2)
ESD/XRT (10)
ESD/XRW (1)

ESD DET 9
APO NY 09021 (2)

Rome Air Development Center
Griffiss AFB, NY 13441
RADC/CC (1)
RADC/CA (2)
RADC/DC (2)
RADC/CO (2)
RADC/OC (2)
RADC/XP (2)

HQ ESC/XOX
Kelly AFB
San Antonio, TX 78243 (2)

Tactical Air Command
Langely AFB, VA 23665
TAFIG/II (2)
TAC/DRC (2)

Tactical Air Command Systems Office
Hanscom AFB, MA 01731
TACSO-E (2)

The MITRE Corporation
Bedford Operations
P.O. Box 208
Bedford, MA 01730
ATTN: Mr Norm Briggs (2)

DARPA/ITPO
1400 Wilson Blvd
Arlington, VA 22209 (1)

USA/CECOM
Ft. Monmouth, NJ 07703 (1)

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314 (12)

AU Library
Maxwell AFB, AL 36112 (1)

NI

DATE